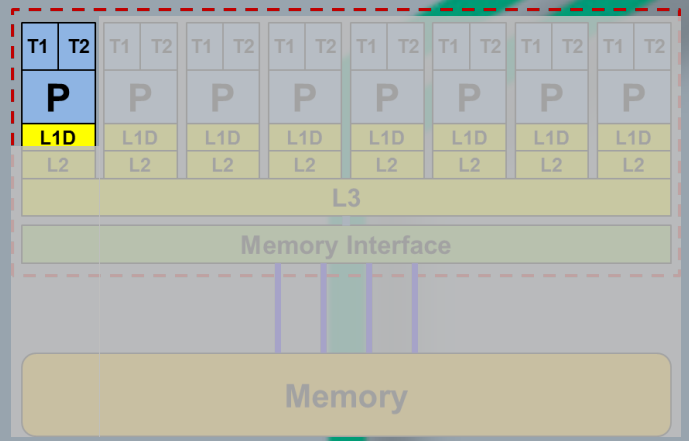
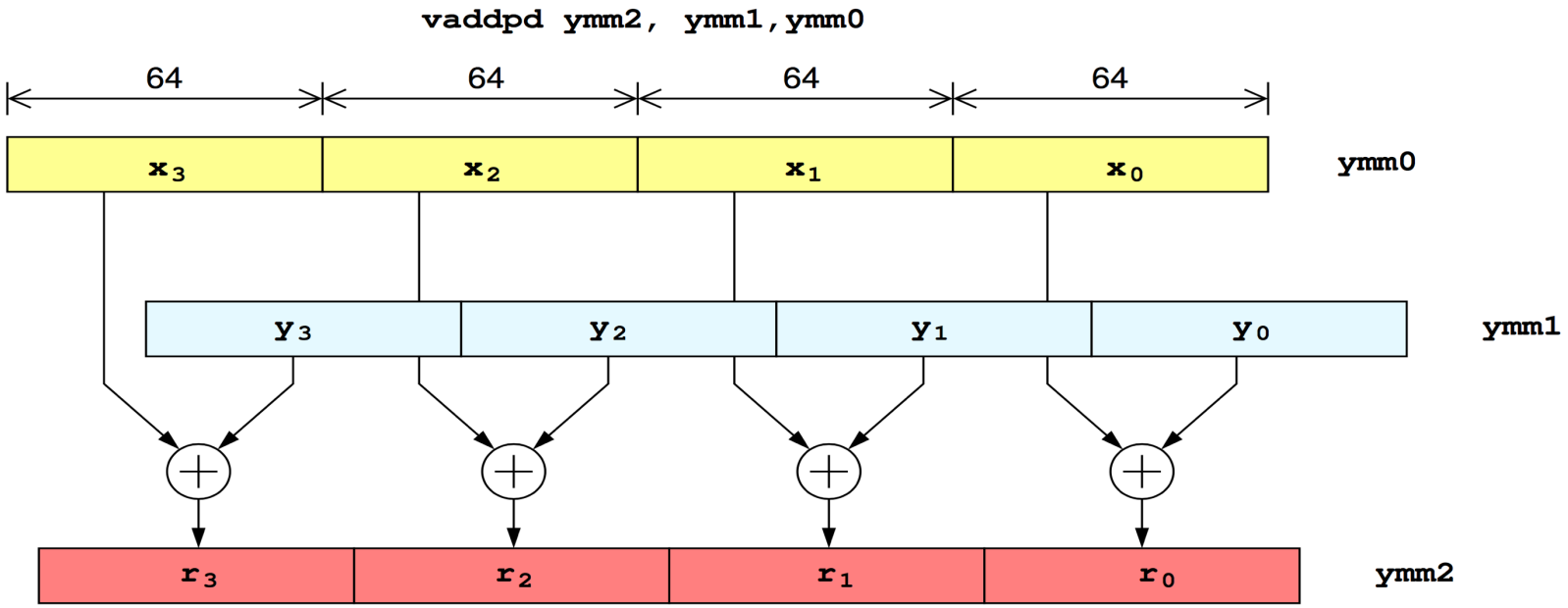
Programming Techniques for Supercomputers
Modern processors: Single Core

- Introduction
  - Basic technology trend / Moore's law
  - Basic concept of core architecture
- Key single core features:
  - Pipelining
  - Superscalarity
  - SingleInstructionMultipleData
- Maximum In-Core Performance



SIMD-processing - Basics

- Apply the same instruction to multiple operands in parallel
- Example: **`vaddpd ymm2, ymm1, ymm0`**
  - Add 4 operands in **`ymm0`** $\{x_0,x_1,x_2,x_3\}$
    to 4 operands in **`ymm1`** $\{y_0,y_1,y_2,y_3\}$ with one instruction
  - "Wide registers" hold multiple operands:
    4 results in **`ymm2`** $\{x_0+y_0, x_1+y_1, x_2+y_2, x_3+y_3\}$



SIMD-processing – Basics

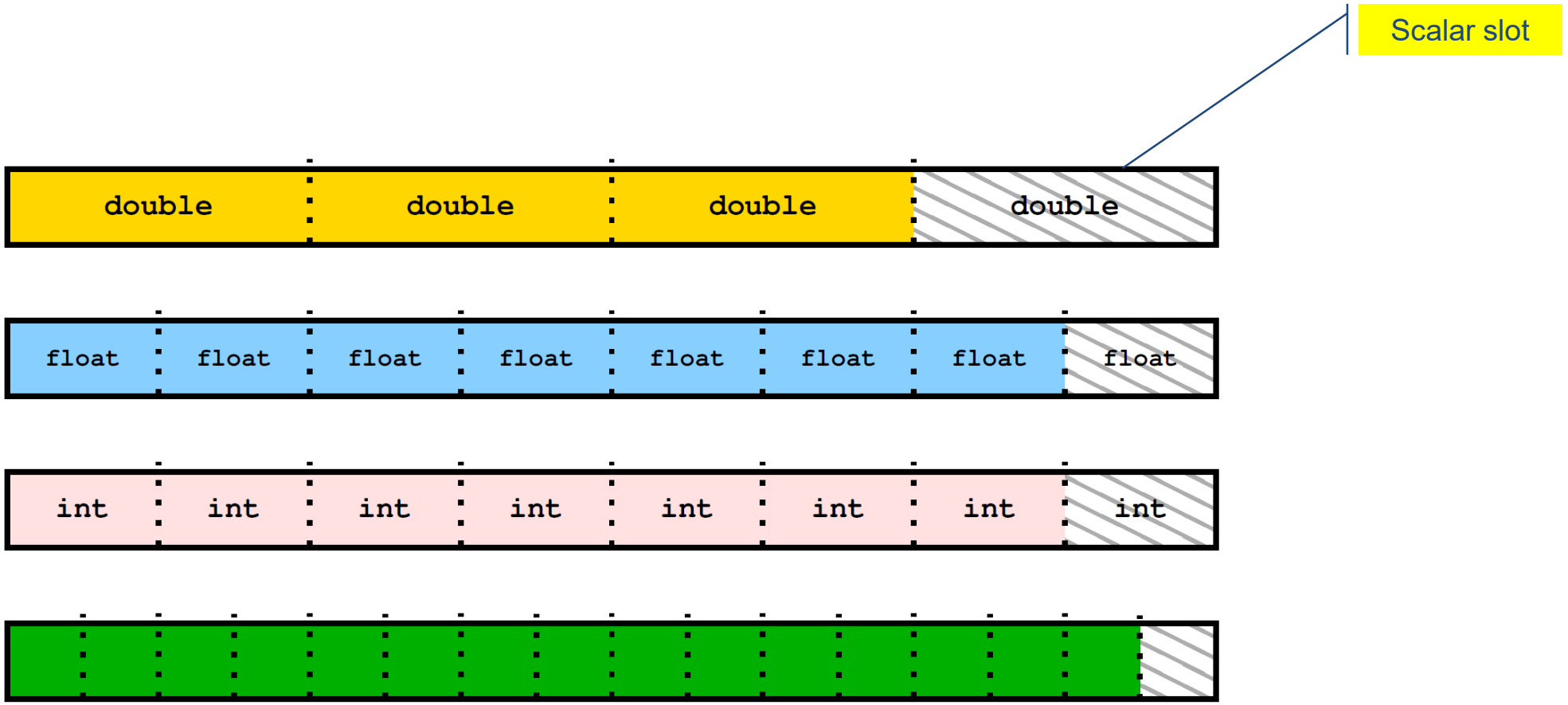
- Single Instruction Multiple Data (SIMD) instructions allow the concurrent execution of the same operation on “wide” registers.

- x86_64 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double (4 single) precision FP operands
  - AVX/AVX2: register width = 256 Bit → 4 double (8 single) precision FP operands
  - AVX512: register width = 512 Bit → 8 double (16 single) precision FP operands
  - “Scalar” (non-SIMD) execution: → 1 double (1 single) precision FP operand
    (only lower 64 bits (32 bit) of registers are used)
  - Integer operands: SSE can be configured very flexible: 1 x 128 bit,…,16 x 8 bit
  - AVX: No support for using the 256 bit register width for integer operations

- SIMD-execution ←→ vector execution
- If compiler has vectorized loop ←→ SIMD instructions are used

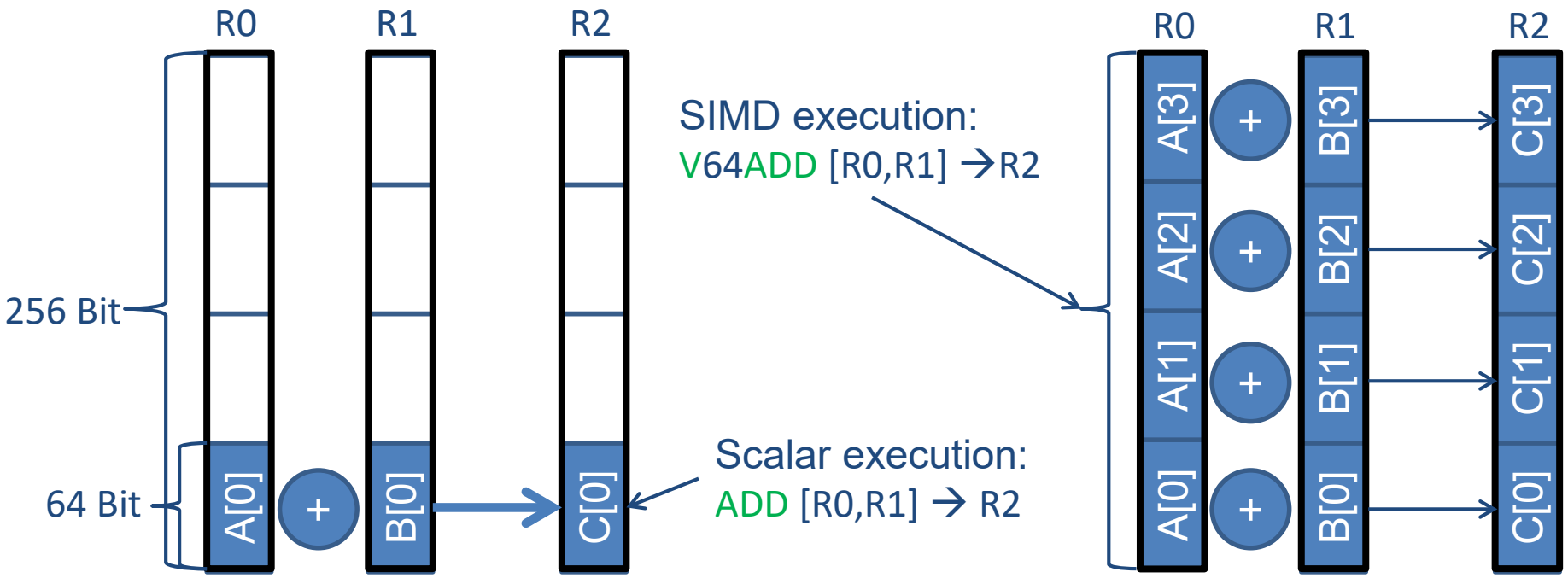
SIMD-processing – Data types in 256 Bit SIMD registers

Supported data types depend on actual SIMD instruction set (starting with AVX2)



SIMD-processing – Basics

- Example: Adding two registers holding double precision floating point operands using 256 Bit register (AVX)



- If 128 Bit SIMD instructions (SSE) are executed only half of the registers width is used

SIMD-processing – Basics

- Steps (done by the compiler) for “SIMD-processing”

```
for(int i=0; i<n;i++)
        C[i]=A[i]+B[i];
```

“Loop unrolling”

```
for(int i=0; i<n;i+=4){
        C[i]  =A[i]  +B[i];
        C[i+1]=A[i+1]+B[i+1];
        C[i+2]=A[i+2]+B[i+2];
        C[i+3]=A[i+3]+B[i+3];}
//remainder loop omitted
```

Don‘t unroll for SIMD yourself – leave it to the compiler!

“Pseudo-Assembler”

Load 256 Bits starting from address of **A[i]** to register **R0**

Add the corresponding 64 Bit entries in **R0** and **R1** and store the 4 results to **R2**

Store **R2** (256 Bit) to address starting at **C[i]**

```
LABEL1:
        VLOAD R0 ← A[i]
        VLOAD R1 ← B[i]
        V64ADD[R0,R1] → R2
        VSTORE R2 → C[i]
        i←i+4
        i<(n-4)? JMP LABEL1
//remainder loop omitted
```

SIMD-processing – Basics

- No SIMD-processing for loops with data dependencies

```
for(int i=0; i<n;i++)
       A[i]=A[i-1]*s;
```

- "Pointer aliasing" may prevent compiler from SIMD-processing

```
void scale_shift(double *A, double *B, double *C, int n) {
       for(int i=0; i<n; ++i)
          C[i] = A[i] + B[i];
}
```

  - C/C++ allows that `A` → `&C[-1]` and `B` → `&C[-2]`
    → `C[i] = C[i-1] + C[i-2]`: dependency → No SIMD-processing

  - If "pointer aliasing" is not used, tell it to the compiler:
    - `-fno-alias` (Intel), `-Msafeptr` (PGI), `-fargument-noalias` (gcc)
    - `restrict` keyword (C99 only!):
      `void f(double restrict *a, double restrict *b) {…}`

SIMD-processing – enabling SIMD / vectorization

- Intel compiler will vectorize starting with **–O2**.
  →To enable specific SIMD extensions use the –x option:
  - **-xSSE2** vectorize for SSE2 capable machines
  - **-xAVX** on Sandy/Ivy Bridge processors
  - **-xCORE-AVX2** on Haswell/Broadwell
  - **-xCORE-AVX512** on Skylake/Cascade Lake and later (certain models)
  - **-xMIC-AVX512** on Xeon Phi Knights Landing
  - **-xHOST** use SIMD extensions of machine you compile on

- Since Intel compiler 12.0 the **simd pragma** is available
  →**#pragma simd** enforces vectorization

- Check if compiler was successful:
  - Check compiler diagnostics:
    -opt-report5
  - Check assembly – if neccessary

```
LOOP BEGIN at Run_BW-Test.F90(278,9)
   remark #25084: Preprocess Loopnests: Moving Out Store    [ Run_BW-Test.F90(280,12) ]
   remark #15388: vectorization support: reference a(in) has aligned access   [ Run_BW-Test.F90(280,26) ]
   remark #15388: vectorization support: reference b(in) has aligned access   [ Run_BW-Test.F90(280,34) ]
   remark #15305: vectorization support: vector length 8
   remark #15399: vectorization support: unroll factor set to 8
   remark #15309: vectorization support: normalized vectorization overhead 0.446
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 7
   remark #15477: vector cost: 0.870
   remark #15478: estimated potential speedup: 6.300
   remark #15488: --- end vector cost summary ---
LOOP END
```

SIMD-processing – Basics

- SIMD-processing of a vector sum

```
s=0.0;
for(int i=0; i<n;i++)
        s = s + A[i];
```

```
s0=0.0;
s1=0.0;
S2=0.0;
S3=0.0;
for(int i=0; i<n;i+=4){
   s0 = s0+ A[i]  ;
   s1 = s1+ A[i+1];
   s2 = s2+ A[i+2];
   s3 = s3+ A[i+3];
}     R0       R1
//remainder
s=s0+s1+s2+s3
```

Data dependency on `s` must be resolved for SIMD-processing (assume AVX)

Compiler does transformation (Modulo Variable Expansion) –

if programmer allows it to do so!
(e.g. use **-O3** instead of **-O1**)

R0 ← (0.d0, 0.d0, 0.d0, 0.d0)

```
…
V64ADD(R0,R1)          → R0
…
```

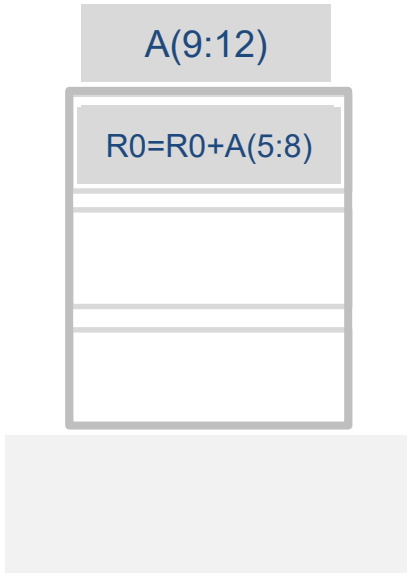
"Horizontal" ADD:
Sum up the 4 64 Bit entries of **R0**

SIMD-processing: What about pipelining?!

```
R0 ←(0.0,0.0,0.0, 0.0)
do i=1, N, 4

   VLOAD A(i:i+3)  → R1
   V64ADD(R0,R1)   → R0

enddo
sum ← HorizontalADD(R0)
```



Need to do another MVE step to fill pipeline stages

“Vertical add”

```
R0 ←(0.0,0.0,0.0,0.0)
R1 ←(0.0,0.0,0.0,0.0)
R2 ←(0.0,0.0,0.0,0.0)
do i=1, N, 12

   VLOAD A(i:i+3)        → R3
   VLOAD A(i+4:i+7)      → R4
   VLOAD A(i+8:i+11)     → R5

   V64ADD(R0,R3)   → R0
   V64ADD(R1,R4)   → R1
   V64ADD(R2,R5)   → R2

enddo
…
V64ADD(R0,R1)      → R0
V64ADD(R0,R2)      → R0

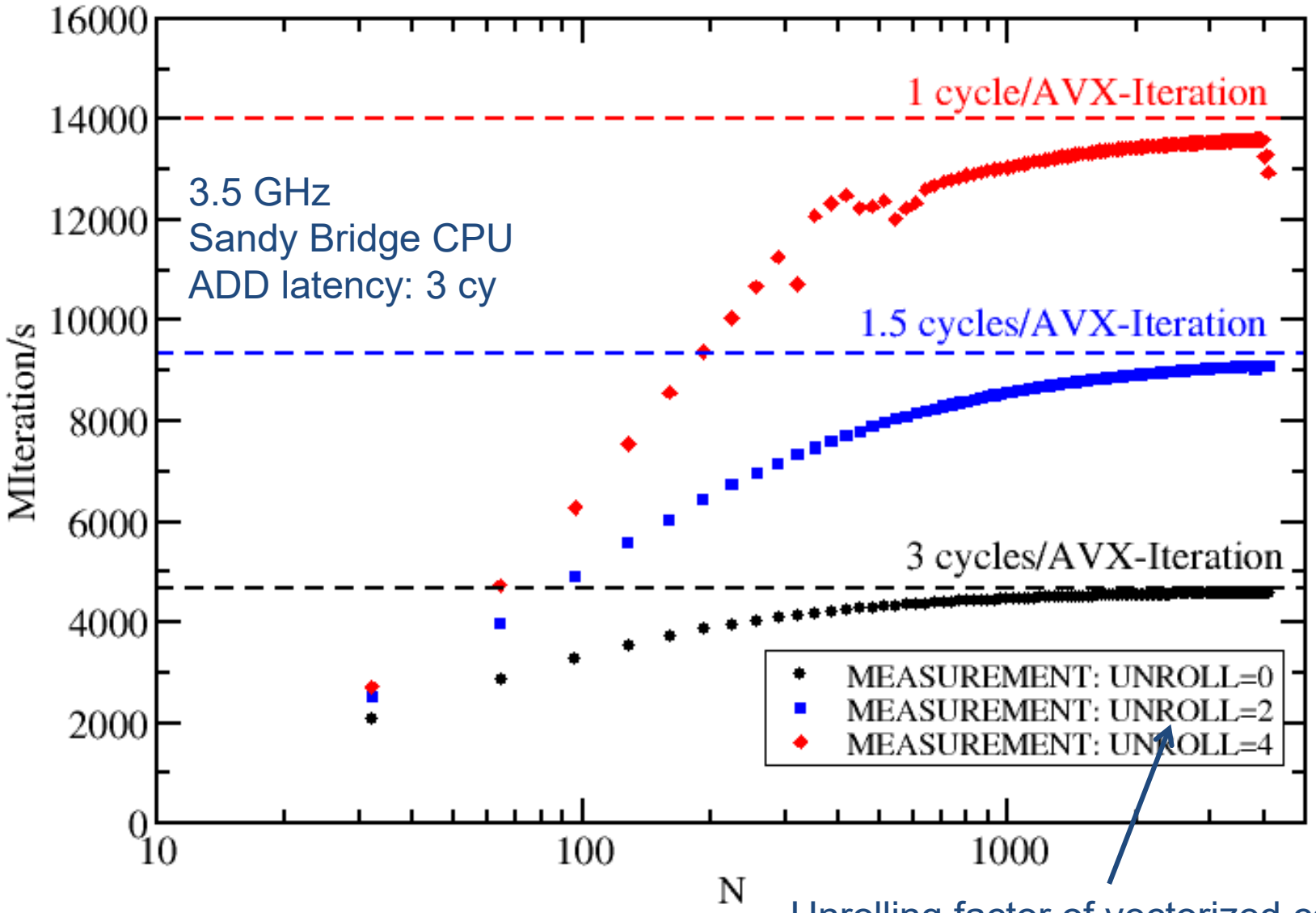
sum← HorizontalADD(R0)
```

SIMD-processing: What about pipelining?!

1 AVX iteration performs 4 i-Iterations (successive)

Performance: 4x higher than “scalar” version

Start-up phase much longer…



SIMD processing – compatible data layouts

Remember:

- SIMD **LD** (**ST**) instructions load (store) mutliple „consecutive“ operands from a baseline adress (from a register) to a register (to a baseline adress)

- SIMD: Apply instruction to vectors (registers) with independent operands

- Which example (data layout) is SIMD compatible?

Structure-of-arrays (SoA)

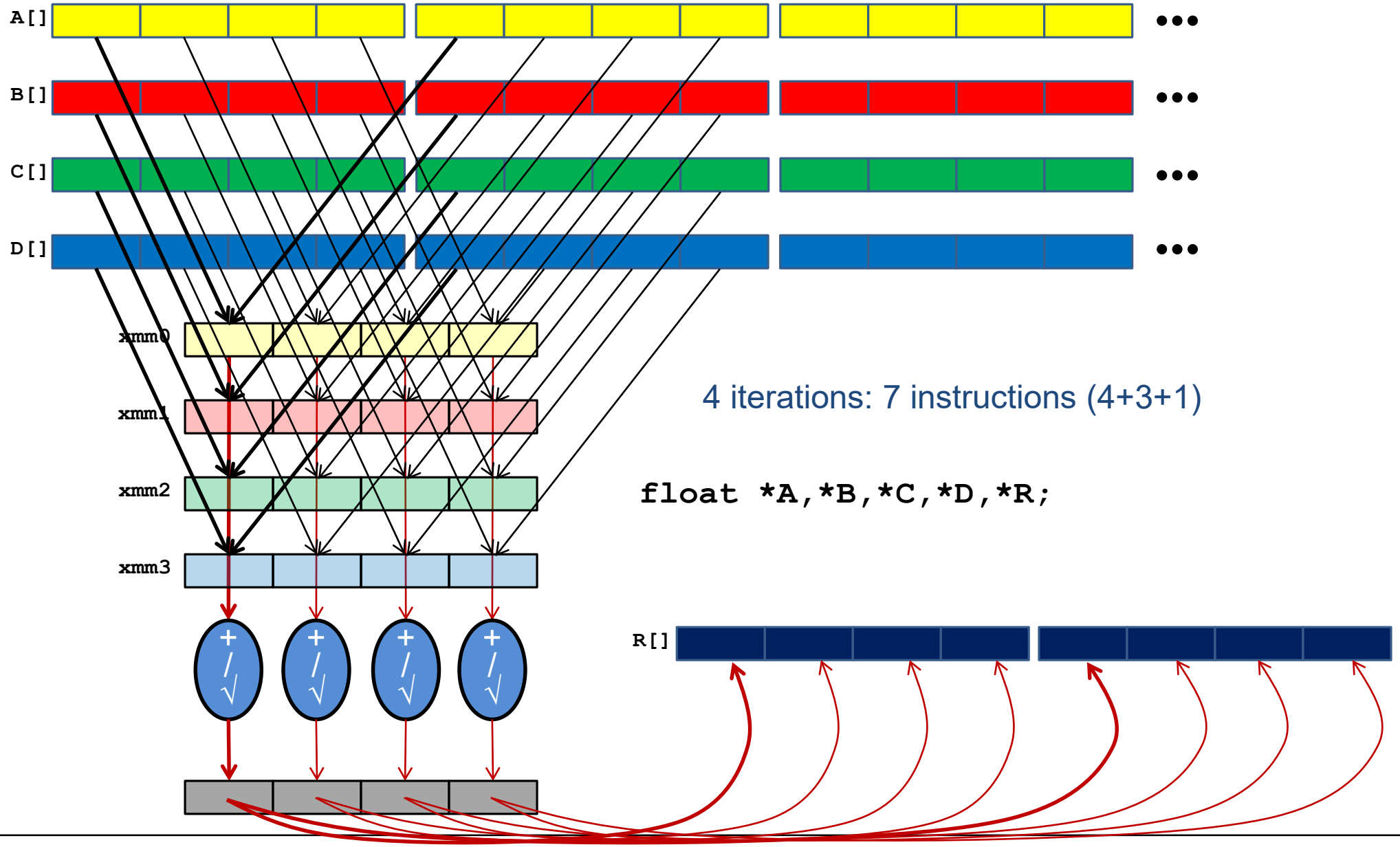
```
…
float *A,*B,*C,*D,*R;
…
for(int i=0; i<n;i++)
 R[i]=((A[i] + C[i]) / B[i])* D[i];
```

Array-of-structures (AoS)

```
…
struct stype {
  float  A,B,C,D,R;
};

struct stype *S;
…
for(int i=0; i<n;i++)
       S[i].R=((S[i].A+S[i].C)/S[i].B)*S[i].D);
```

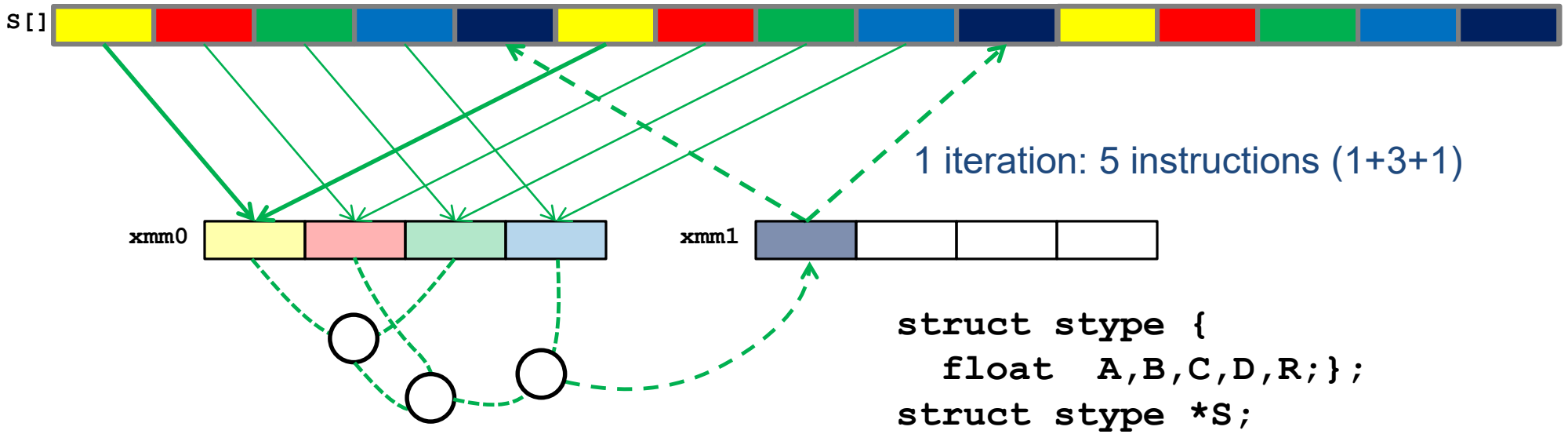
SIMD-processing – data layout: structure-of-arrays (SoA)



4 iterations: 7 instructions (4+3+1)

```
float *A,*B,*C,*D,*R;
```

SIMD-processing – data layout: Array-of-structures (AoS)



Things to watch out for:

- LDs, STs, arithmetic:
  all instructions need to be SIMD-parallel
- “Horizontal” operations across slots of a register are slow (no good support in instruction sets)
- A (limited) number of parallel data streams are not a problem (5 previous slide)
- Carefully check your data layout – e.g. AoS vs. SoA

SIMD-processing – basic rules

Rules for vectorizable loops

1. Inner loop
2. Countable (loop length can be determined at loop entry)
3. Single entry and single exit
4. Straight line code (no conditionals)
5. No (unresolvable) read-after-write data dependencies
6. No function calls (exception intrinsic math functions)

Better performance with:

1. Simple inner loops with unit stride (contiguous data access)
2. SIMD-compatible data layout
3. Minimize indirect addressing
4. Align data structures to SIMD width boundary
5. In C use the `restrict` keyword and/or `const` qualifiers and/or compiler options to rule out array/pointer aliasing

Vectorizability (or lack thereof)s

- Non-countable loop
  - See https://software.intel.com/en-us/node/522574

```
void no_cnt(float a[], float b[], float c[]) {
  int i=0; // Iterations dependent on a[i].
  while (a[i]>0.0) {
    a[i] = b[i] * c[i];
    i++;
  }
}
```

- Loop with non-resolvable branch

```
void exit_two(float a[], float b[], float c[], int n) {
  for(int i=0; i<n; ++i) {
    a[i] = b[i] * c[i];
    if(a[i]<0) {
      printf("WARNING!");  // remainder from debugging
      break;
}}}
```

Costs of arithmetic operations: Intel Skylake processors

From instructions to FP operations:

1. (Scalar) FMA instruction: 2 FP operations (MULT&ADD)
2. SIMD Instructions: Consider register width!

	Latency [cy/instruction]	Throughput [FP operations/cy]		
		Scalar	AVX	AVX512
MULT DP	4	2	8	16
FMA DP	4	4	16	32
FMA SP	4	4	32	64
SQRT DP	26	1/12 = 0.083	1/3	1/3
SQRT SP	20	1/6 = 0.16	2/3	2/3
DIV DP	14	1/4 = 0.25	1/2	1/2

→ SQRT operation may be approx. 50x more expensive than MULT operation!
(See https://arxiv.org/abs/1702.07554 for older Intel architectures)

Modern Processors – Single Core Summary (1)

- Pipelining & Superscalarity: Exploits Instruction Level Parallelism
  - Obstacles:
    - Instruction / Loop carried dependencies
    - Latencies
  - Enabling Techniques
    - Software Pipelining & Loop unrolling (compile time)
    - Out-of-order Execution (runtime)
    - Simultaneous Multithreading (runtime & compile) – see later

- Performance limit – Pipelining: 1 instruction / cycle

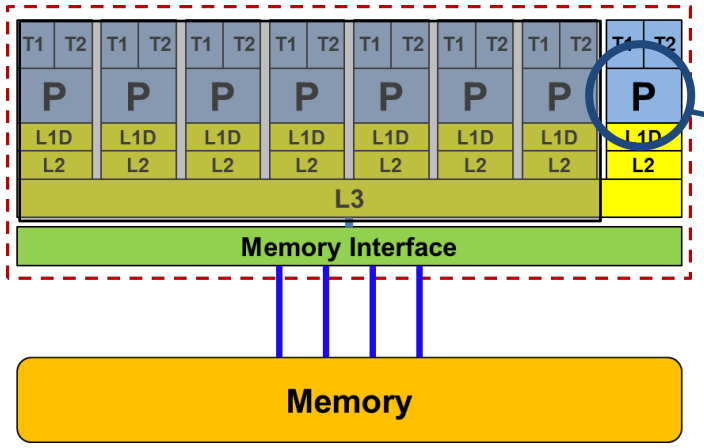
- Performance limit – Superscalarity: n_{super} instructions / cycle
  (e.g. for FP instructions $n_{super} \leq 2$ for Intel x86)

- Consider type of basic FP instructions (n_{FMA}):
  FMA ($n_{FMA} = 2$) or MULT/ADD ($n_{FMA} = 1$)

Modern Processors – Single Core Summary (2)

- Single Instruction Multiple Data (SIMD): Exploits Data Parallelism
  - Obstacles:
    - Loop carried dependencies
    - Non-vectorizable parts (e.g. non-consecutive data access)
  - Enabling Techniques
    - (Inner) Loop Unrolling + packed SIMD instructions (Compile Time)
- Performance limit – SIMD: n_{SIMD} operations / instruction (e.g. n_{SIMD} =4 for double precision FP MULT/ADD with AVX & AVX2)

There is no single driving force for single core performance!



Peak Floating Point (FP) Performance:

$$P_{core} = n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$$

Super-scalarity

FMA factor

SIMD factor

Clock Speed

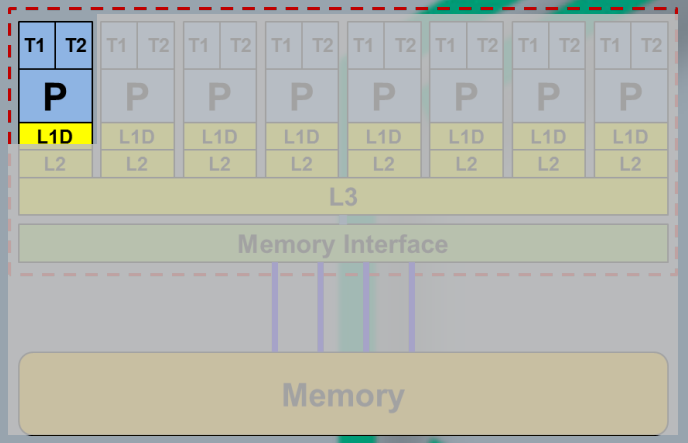
Assume 64-bit operands	n_{super}^{FP} [inst./cy]	n_{FMA}	n_{SIMD} [FPops/inst.]		Server	f [GHz]	P_{core}[GF/s]
Nehalem	2	1	2	Q1/2009	X5570	2.93	11.7
Westmere	2	1	2	Q1/2010	X5650	2.66	10.6
Sandy Bridge	2	1	4	Q1/2012	E5-2680	2.7	21.6
Ivy Bridge	2	1	4	Q3/2013	E5-2660 v2	2.2	17.6
Haswell	2	2	4	Q3/2014	E5-2695 v3	2.3	36.8
Broadwell	2	2	4	Q1/2016	E5-2699 v4	2.2	35.2
Skylake	2	2	8	Q3/2017	Gold 6148	2.4	76.8
AMD Zen2	2	2	4	Q3/2019	Epyc 7742	2.25	36.0
Fujitsu A64FX	2	2	8	Q2/2020	FX700	1.8	57.6
IBM POWER8	2	2	2	Q2/2014	S822LC	2.93	23.4

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Programming Techniques for Supercomputers
Modern processors: Single Core

- Introduction
  - Basic technology trend / Moore's law
  - Basic concept of core architecture
- Key single core features:
  - Pipelining
  - Superscalarity
  - SingleInstructionMultipleData
- Maximum In-Core Performance



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Maximum In-Core Performance P_{max}: Upper performance bound based on instruction execution



Calculating P_{max} - Idea

- Estimate minimum time [cycles] to execute a given set of instructions on a give single core processor

- Given
  - „Application Model“: Set of instructions (including potent. dependencies):
  - „Machine Model“: Microarchitectural information, e.g. Instruction throughputs and latencies

→ Minimum execution time

→ Maximum (attainable) performance (P_{max}) for given set of instructions

→ $P_{max} \leq P_{core}$

Calculating P_{max} - basic assumptions

- P_{max} ←→ instruction execution

Cycle 10	load a[10]	add a[6]=c,a[6]	store a[3]
Cycle 11	load a[11]	add a[7]=c,a[7]	store a[4]
Cycle 12	load a[12]	add a[8]=c,a[8]	store a[5]
Cycle 13	load a[13]	add a[9]=c,a[9]	store a[6]
Cycle 14	load a[14]	add a[10]=c,a[10]	store a[7]
Cycle 15	load a[15]	add a[11]=c,a[11]	store a[8]

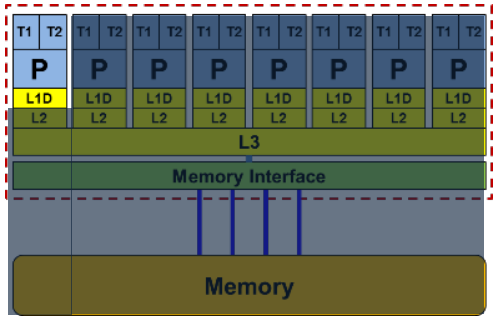
- Assumptions
  - Steady state execution (no start-up latencies)
  - Perfect re-ordering of instructions
  - Only loop-carried dependencies (`a(i)=a(i-1)*s`) are taken into account;
  - Consider only LD/ST/MULT/ADD/FMA instructions
  - All data in L1 cache

- Determine
  - Basic unit of work (W: e.g. 1,2,4 or 8 loop iterations) and instruction mix to execute it → map instruction mix to hardware units/ports
  - Execution unit / port which requires longest to execute the instructions mapped to it („bottleneck“) and its utilization time → $T_{max}^{inst}\ [cy]$
  - Performance $P_{max} = {}^{W}/_{T_{max}^{inst}}\ [\frac{work}{cy}]$ – multiply with clock speed (if required)

Calculating P_{max} - basic machine model

Intel Xeon E5 multicore processors –LD/ST/MULT/ADD/FMA throughput

Microarchitecture	SandyBridge-EP	IvyBridge-EP	Haswell-EP
Shorthand	SNB	IVB	HSW
Xeon Model	E5-2680	E5-2690 v2	E5-2695 v3
Year	03/2012	09/2013	09/2014
Clock speed (fixed)	2.7 GHz	2.2 GHz	2.3 GHz
Cores/Threads	8/16	10/20	14/28
Load/Store throughput per cycle			
AVX(2)	1 LD & 1/2 ST	1 LD & 1/2 ST	2 LD & 1 ST
SSE/scalar	2 LD ‖ 1 LD & 1 ST	2 LD ‖ 1 LD & 1 ST	2 LD & 1 ST
L1 port width	2×16+1×16 B	2×16+1×16 B	2×32+1×32 B
ADD throughput	1 / cy	1 / cy	1 / cy
MUL throughput	1 / cy	1 / cy	2 / cy
FMA throughput	n/a	n/a	2 / cy



LOAD/STORE instruction throughput / cy

FP instructions throughput – max: 2 / cy

1 LD/cy & 1 ST/2cy with AVX

2 LD/cy OR (1LD+1ST)/cy with SSE or scalar

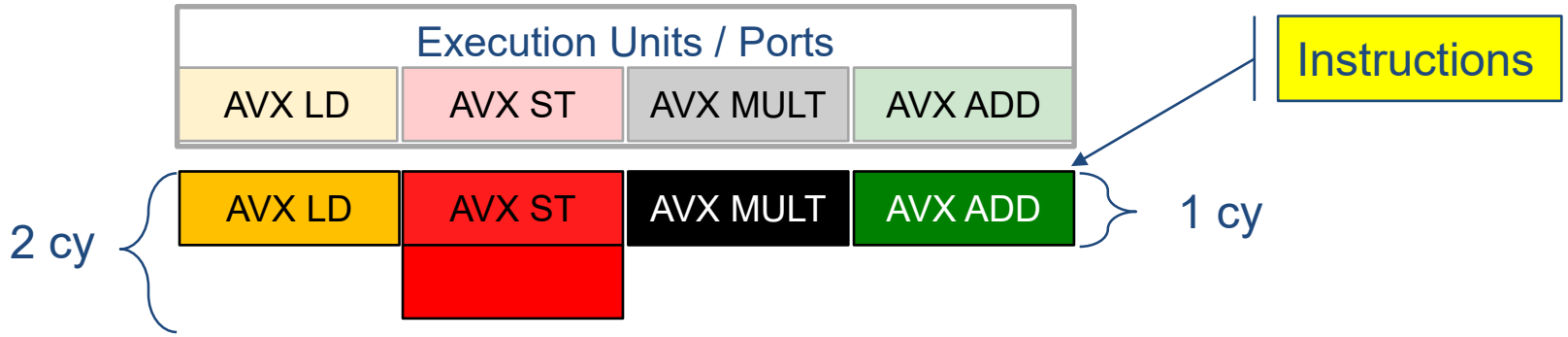
2 FMA/cy

Typically a max. of 4 instructions per cycle can be executed, i.e. $CPI \geq 0.25 \frac{cy}{inst}$.

Calculating P_{max} - basic machine model / execution ports

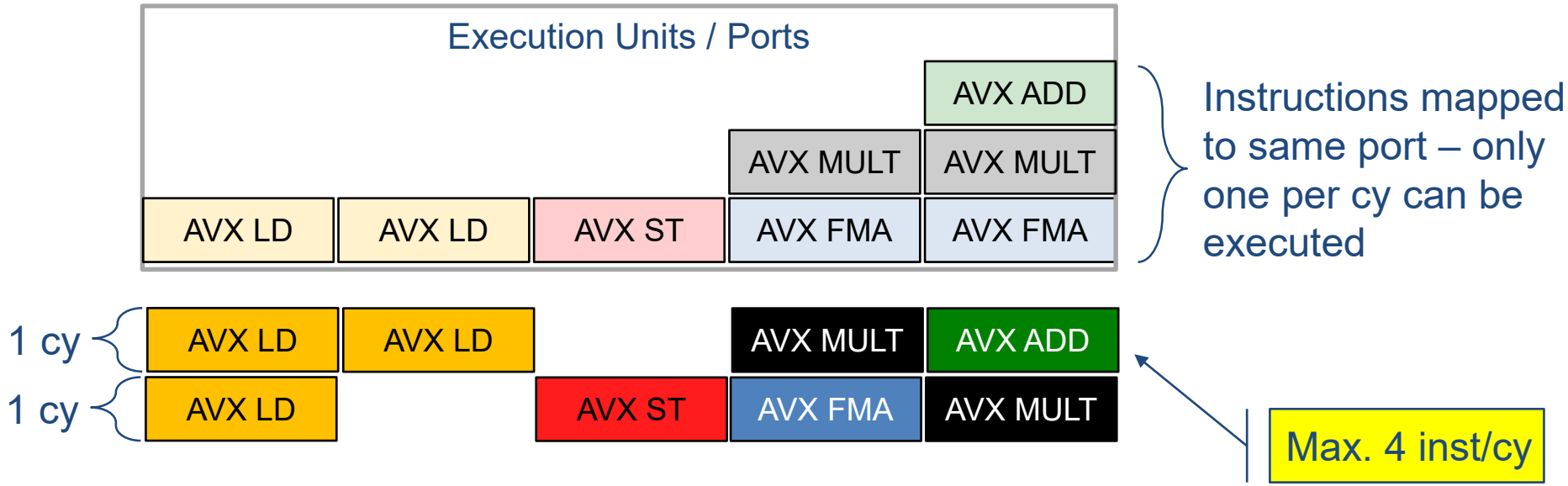
Instruction mix
1 AVX LD
1 AVX ST
1 AVX MULT
1 AVX ADD

Intel Sandy Bridge / Ivy Bridge



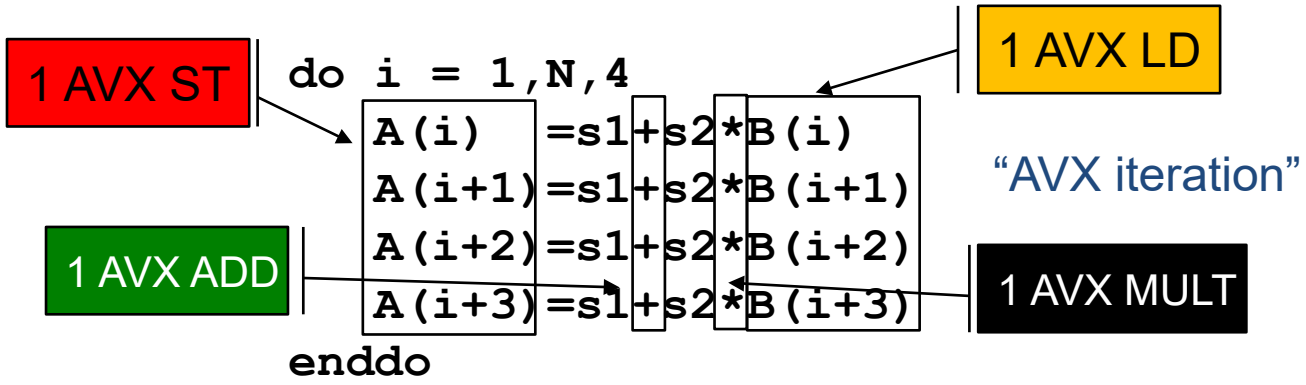
Intel Haswell / Broadwell

Instruction mix
3 AVX LD
1 AVX ST
2 AVX MULT
1 AVX ADD
1 AVX FMA

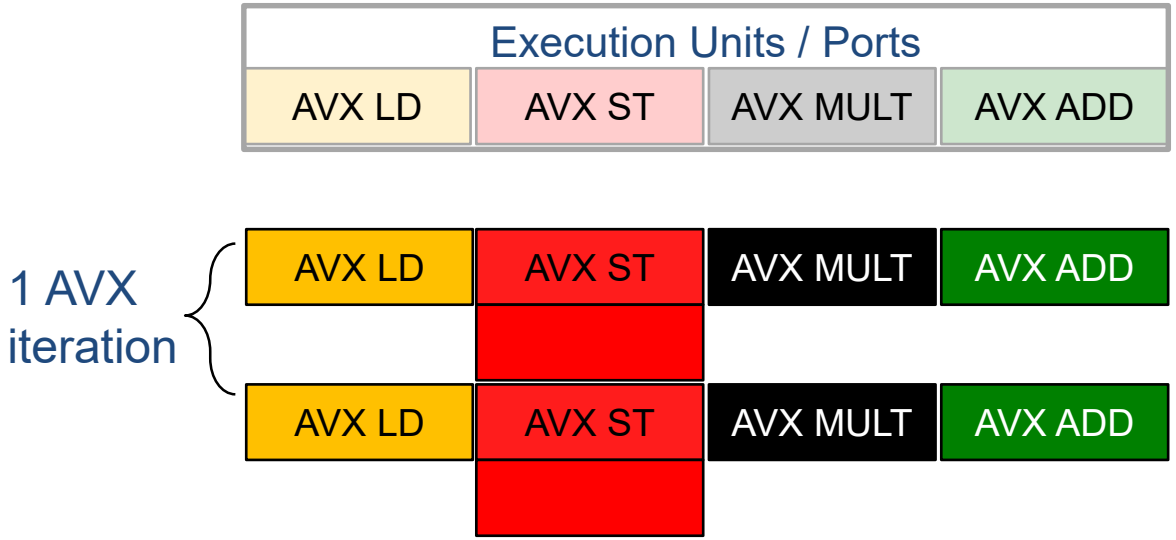


Calculating P_{max} – example (1): `A(i)=s1+s2*B(i)`

```
do i = 1,N
   A(i)=s1+s2*B(i)
enddo
```



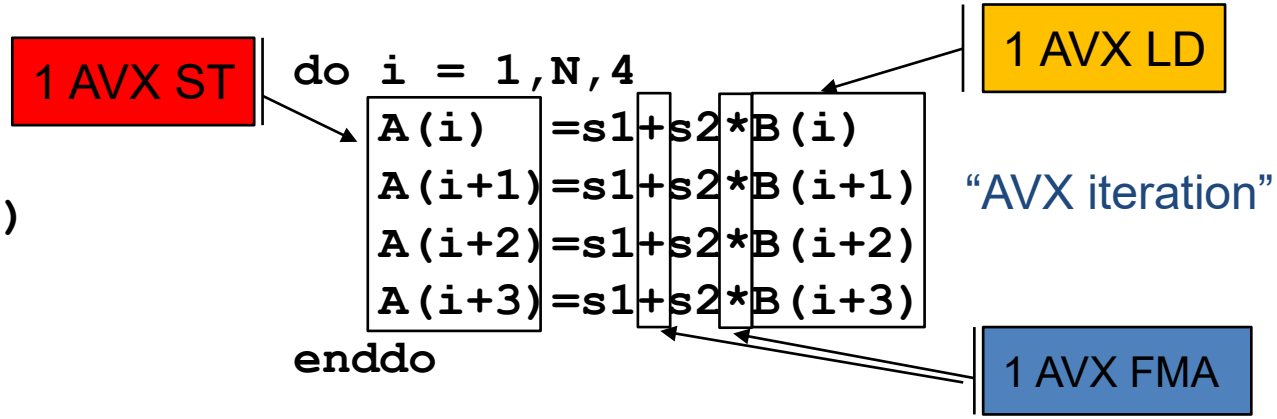
AVX performance (64 Bit) on Sandy Bridge/Ivy Bridge:



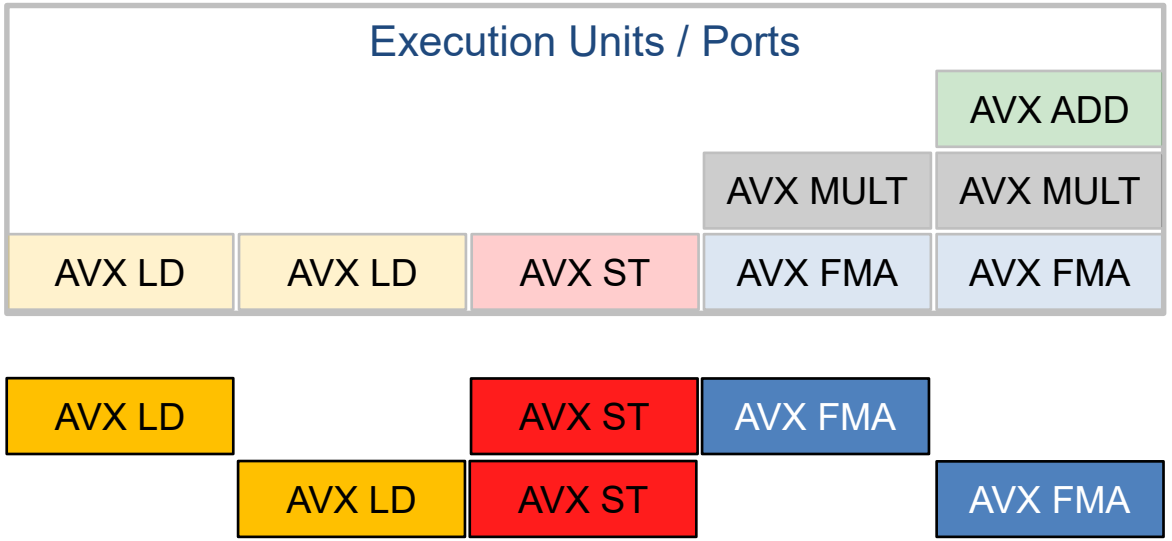
- Bottleneck: STore unit
- 1 AVX iteration: $T_{max}^{inst} = 2cy$
- 1 AVX iteration
  → 4 loop iterations → 8 F
- $P_{max} = {}^{8F}/_{2cy} = 4\ {}^{F}/_{cy}$

Calculating P_{max} – example (1): `A(i)=s1+s2*B(i)`

```
do i = 1,N
   A(i)=s1+s2*B(i)
enddo
```



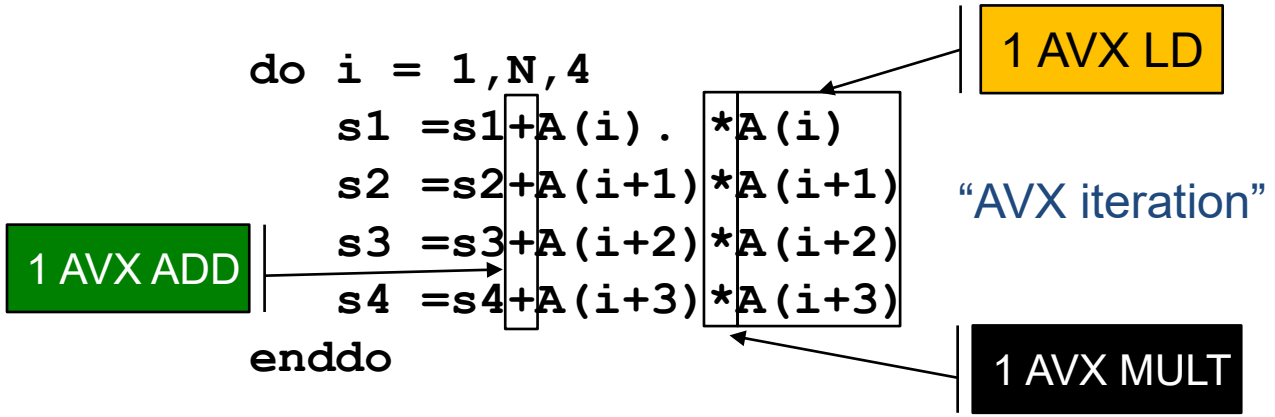
AVX performance (64 Bit) on Haswell / Broadwell



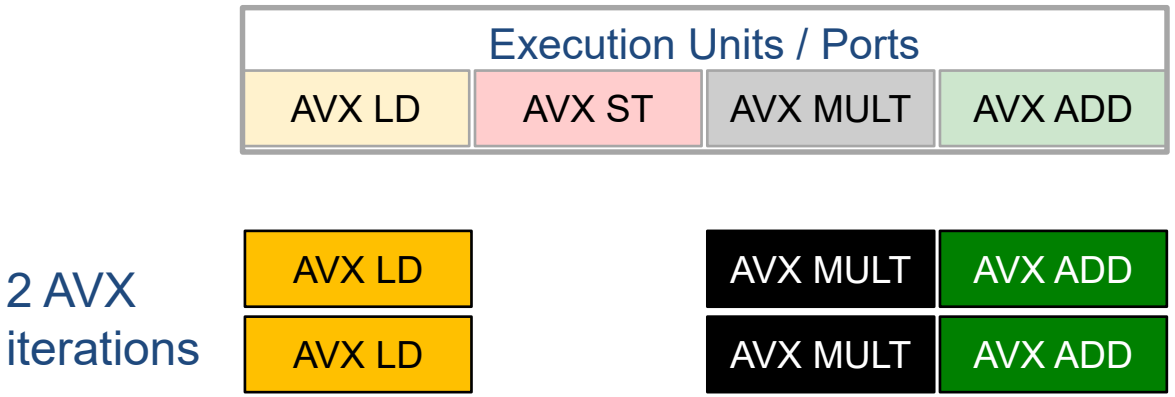
- Bottleneck: ST
- 1 AVX iteration: $T_{max}^{inst} = 1cy$
- 1 AVX iteration
  → 4 loop iterations → 8 F
- $P_{max} = {}^{8F}/_{1cy} = 8\ {}^{F}/_{cy}$

Calculating P_{max} – example (2): `s=s+A(i)*A(i)`

```
do i = 1,N
   s=s+A(i)*A(i)
enddo
```



AVX performance on Sandy Bridge/Ivy Bridge



- Bottleneck: LD/MULT/ADD
- 1 AVX iteration: $T_{max}^{inst} = 1cy$
- 1 AVX iteration
  → 4 loop iterations → 8 F
- $P_{max} = {}^{8F}/_{1cy} = 8\ {}^{F}/_{cy}$

Calculating P_{max} – example (2): `s=s+A(i)*A(i)`

```
do i = 1,N
   s=s+A(i)*A(i)
enddo
```

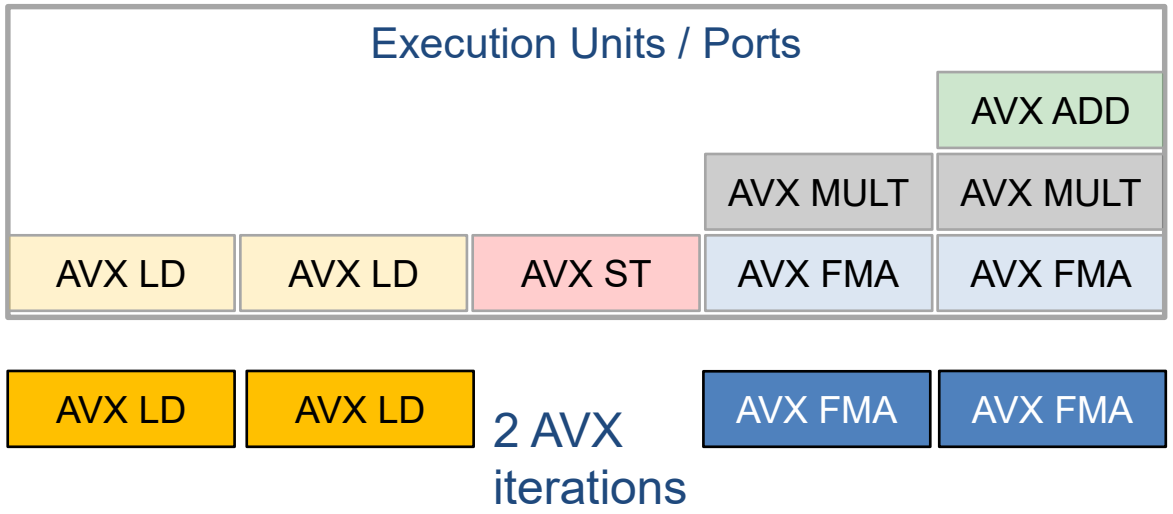
```
do i = 1,N,4
   s1 =s1+A(i).  *A(i)
   s2 =s2+A(i+1)*A(i+1)
   s3 =s3+A(i+2)*A(i+2)
   s4 =s4+A(i+3)*A(i+3)
enddo
```

1 AVX LD

"AVX iteration"

1 AVX FMA

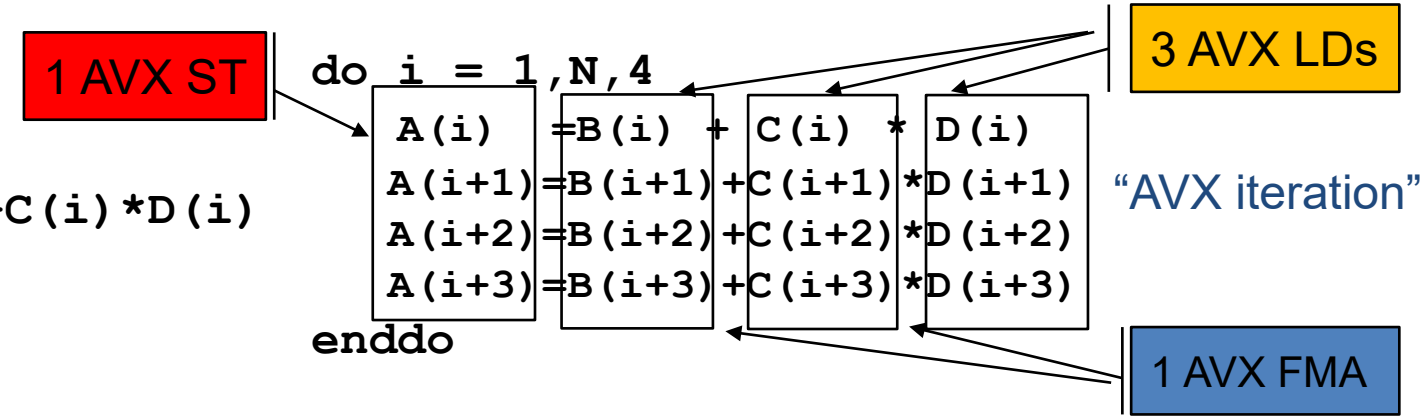
AVX performance on Haswell / Broadwell



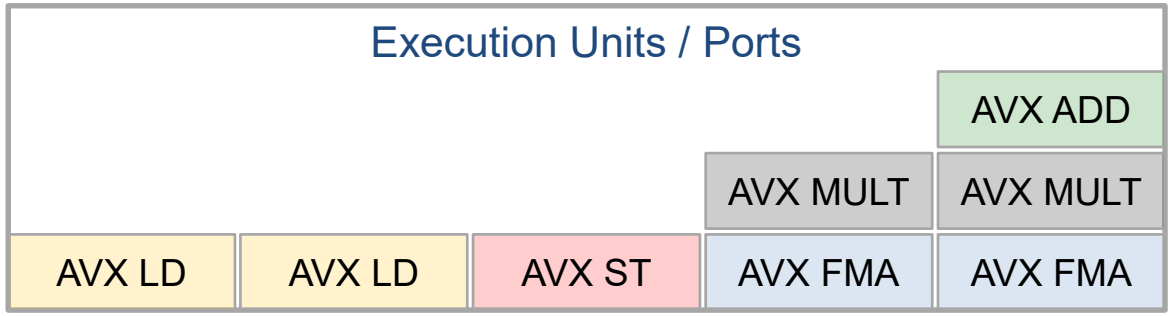
- Bottleneck: LD/FMA
- 2 AVX iteration: $T_{max}^{inst} = 1cy$
- 2 AVX iteration
  → 8 loop iterations → 16 F
- $P_{max} = {}^{16F}/_{1cy} = 16\ {}^{F}/_{cy}$

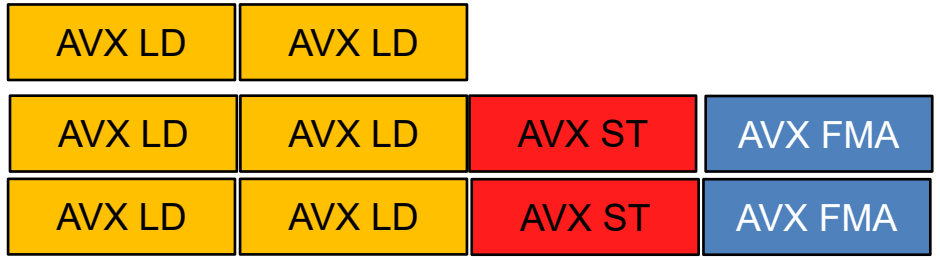
Calculating P_{max} – example (3): `A(i)= B(i)+C(i)*D(i)`

```
do i = 1,N
   A(i)=B(i)+C(i)*D(i)
enddo
```



AVX performance on Haswell / Broadwell

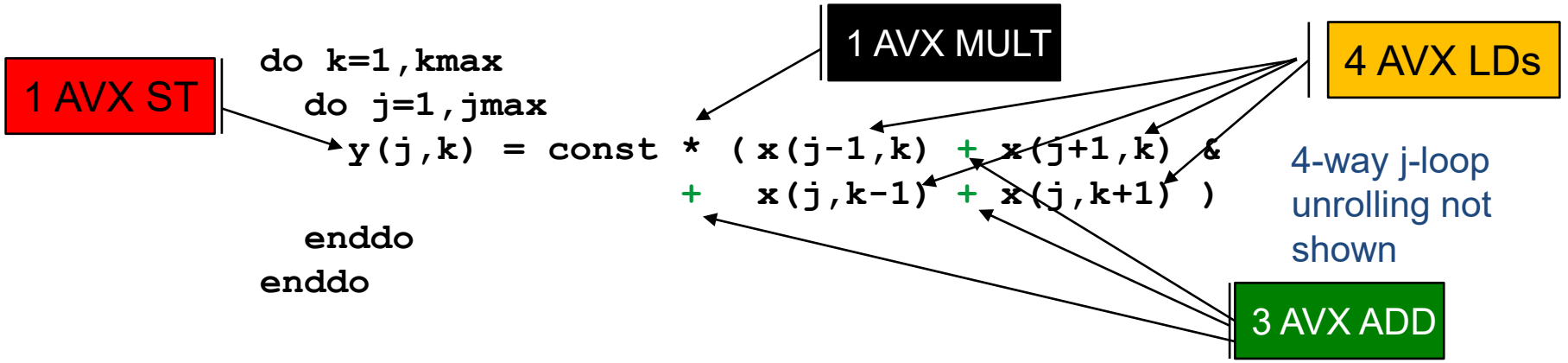




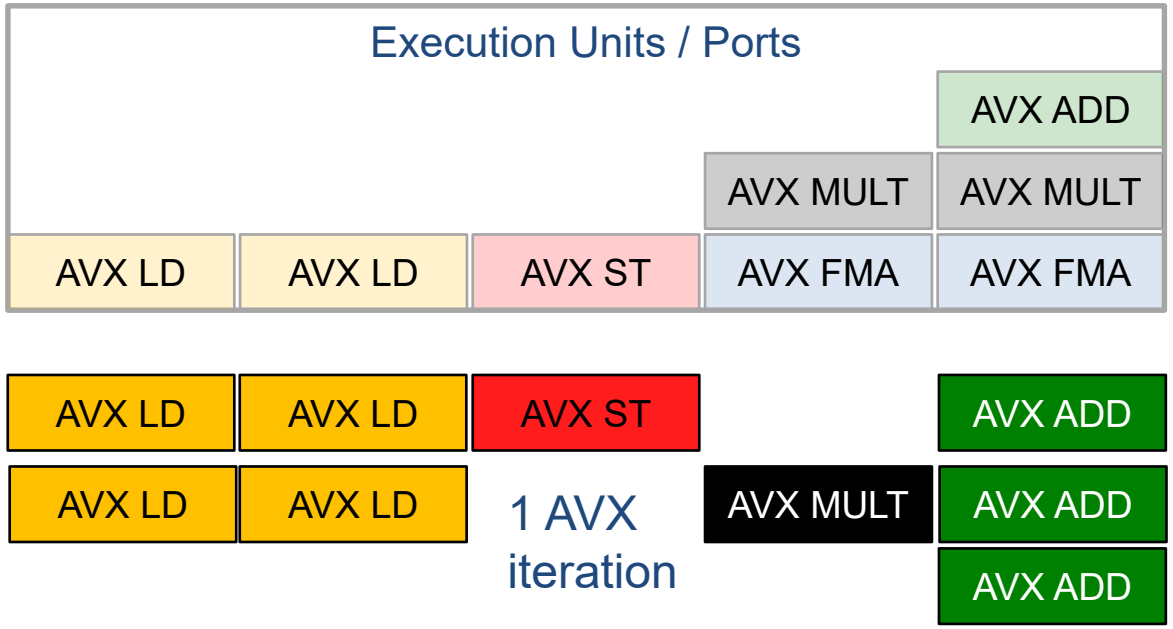
2 AVX iterations

- Bottleneck: LD
- 2 AVX iteration: $T_{max}^{inst} = 3cy$
- 2 AVX iteration → 8 loop iterations → 16 F
- $P_{max} = {}^{16F}/_{3cy} = 5.33\ {}^{F}/_{cy}$

Calculating P_{max} – example (4): 2D 5pt stencil



AVX performance on Haswell / Broadwell



- Bottleneck: ADD
- 1 AVX iteration: $T_{max}^{inst} = 3cy$
- 1 AVX iteration
  → 4 loop iterations → 16 F
- $P_{max} = {}^{16F}/_{3cy} = 5.33\ {}^{F}/_{cy}$

(equivalent:
$P_{max} = {}^{4\ iterations}/_{3cy}$)

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Advanced issues – Intel Cascade Lake / Ice Lake Platinum

Some notes on more recent processors including Intel Xeon 8360Y (Fritz)



Microarchitecture	Ivy Bridge EP	Broadwell EP	Cascade Lake SP	Ice Lake SP
Introduced	09/2013	03/2016	04/2019	06/2021
Cores	≤ 12	≤ 22	≤ 28	≤ 40
LD/ST throughput per cy:				
AVX(2) / AVX512	**1 LD + ½ ST**	**2 LD + 1 ST**	**2 LD + 1 ST**	**2 LD + 1 || 2 ST***
SSE / scalar	**2 LD || 1 LD & 1 ST**			
ADD throughput	**1 / cy**	**1 / cy**	**2 / cy**	**2 / cy**
MUL throughput	**1 / cy**	**2 / cy**	**2 / cy**	**2 / cy**
FMA throughput	**N/A**	**2 / cy**	**2 / cy**	**2 / cy**
L1-L2 data bus	32 B/cy	64 B/cy	64 B/cy	64 B/cy
L2-L3 data bus	32 B/cy	32 B/cy	16+16 B/cy	16+16 B/cy
L1/L2 per core	32 KiB / 256 KiB	32 KiB / 256 KiB	32 KiB / 1 MiB	48 KiB / 1.25 MiB
LLC	2.5 MiB/core inclusive	2.5 MiB/core inclusive	1.375 MiB/core exclusive/victim	1.5 MiB/core exclusive/victim
Memory	4ch DDR3	4ch DDR3	6ch DDR4	8ch DDR4
Memory BW (meas.)	~ 48 GB/s	~ 62 GB/s	~ 115 GB/s	~ 160 GB/s

Source: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html

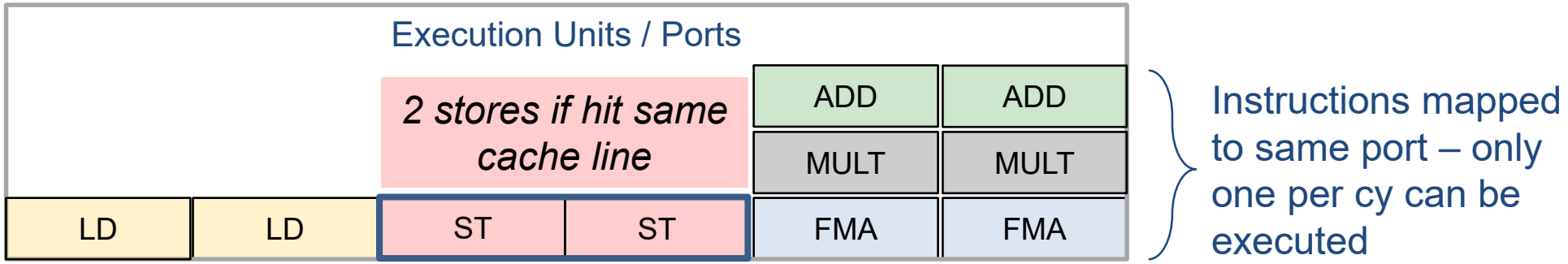
*ICL: 1 ST for AVX512; 2 ST for AVX(2)/SSE/scalar if same cache line

Calculating P_{max} - basic machine model / Icelake (scalar instr.)

Intel Icelake – scalar code

32/64-Bit instruction

Intel Xeon 8360Y – base clock: 2.4 GHz (max. turbo: 3.5 GHz)



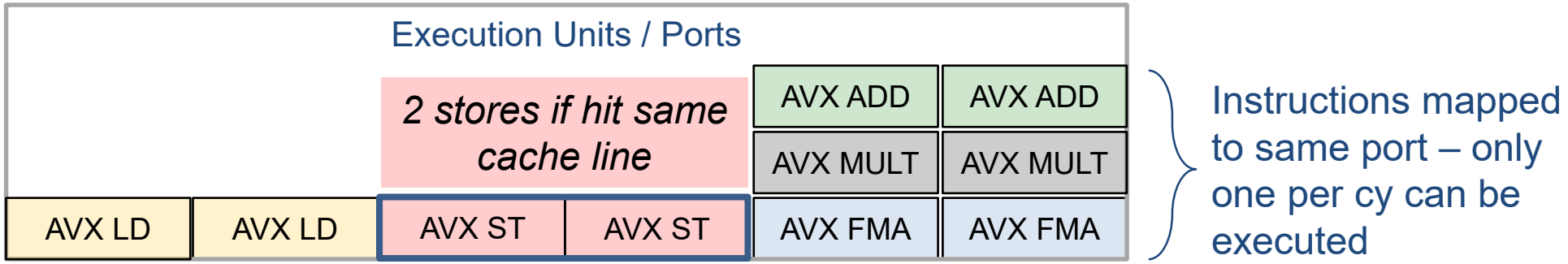
This core has a max. of **5 instructions per cycle** can be executed, i.e. $CPI \geq 0.20 \frac{cy}{inst}$.

Calculating P_{max} - basic machine model / Icelake (AVX instr.)

Intel Icelake – AVX code

256-Bit instruction

Intel Xeon 8360Y – base clock: 2.1 GHz (max. turbo: 3.4 GHz)



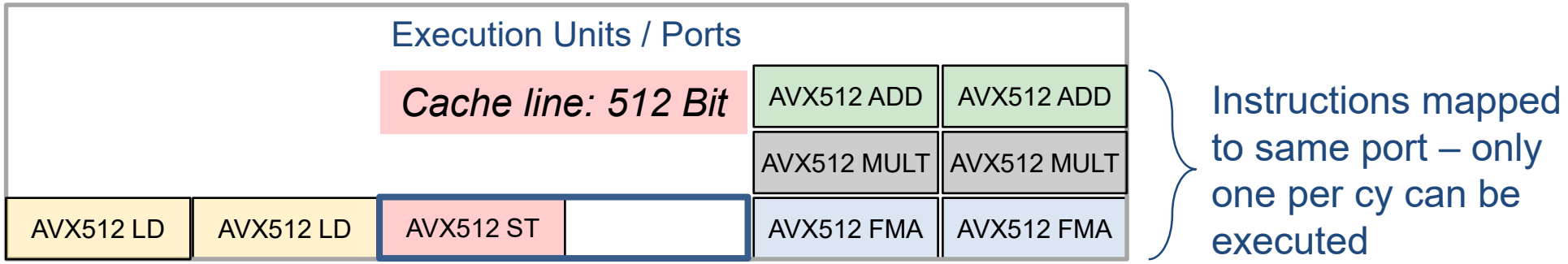
This core has a max. of **5 instructions per cycle** can be executed, i.e. $CPI \geq 0.20 \frac{cy}{inst}$.

Calculating P_{max} - basic machine model / Icelake (AVX instr.)

Intel Ice Lake – AVX512 code

512-Bit instruction

Intel Xeon 8360Y – base clock: 1.8 GHz (max. turbo: 3.4 GHz)



Cascade Lake (same machine model for AVX512)

This core has a max. of **5 instructions per cycle** can be executed, i.e. $CPI \geq 0.20 \frac{cy}{inst}$.

Calculating P_{max} - basic machine model / Ice Lake (scalar instr.)

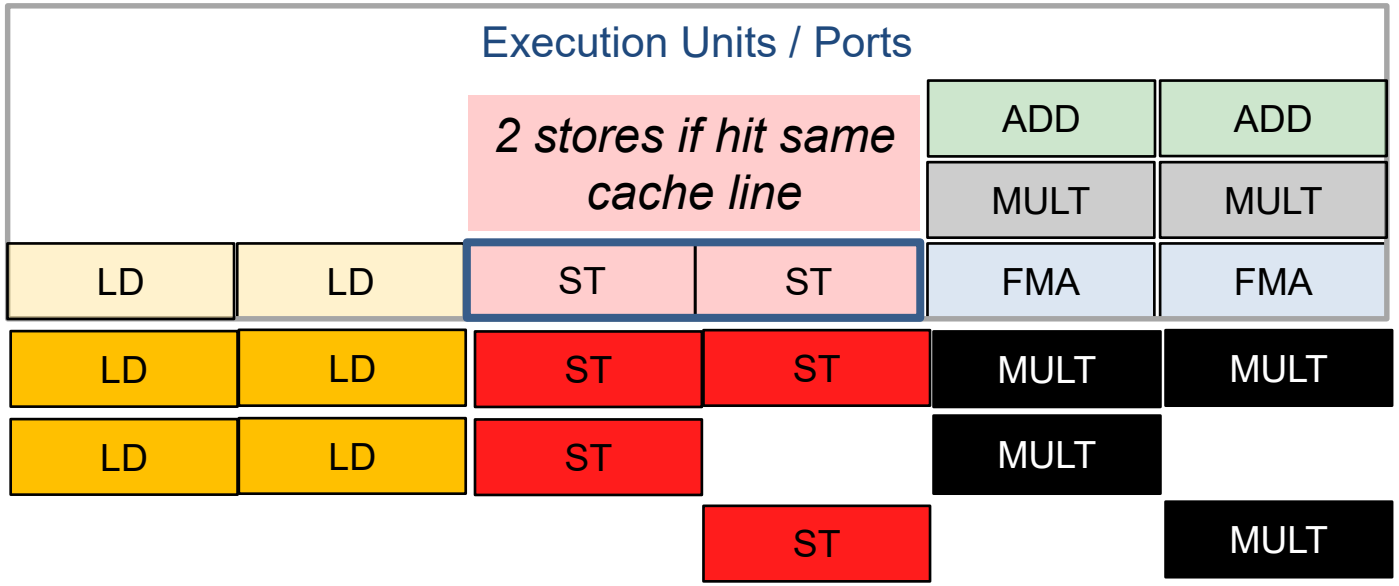
Intel Ice Lake – scalar code: 32/64-Bit instruction

```
do i = 1,N
   A(i)= A(i)*s
enddo
```

1 LD

1 MULT

1 ST



Best performance: 2 MULT in $\frac{6\ inst.}{5\ inst./cy} = 1.2\ cy$ → 0.6 cy. / MULT → 3.33 GF/s (@2GHz)

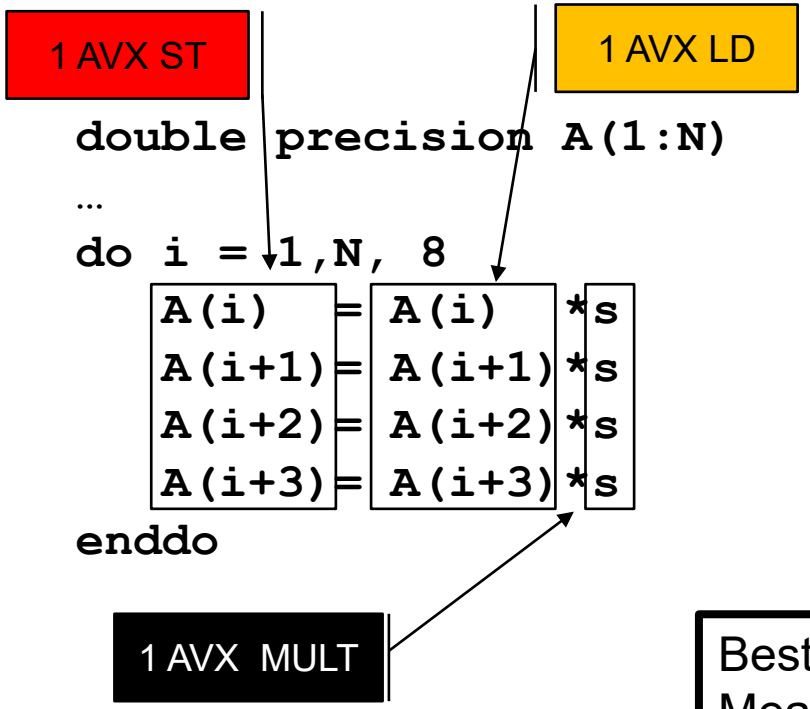
Measured performance: 0.7 cy. / MULT → 2.85 GF/s (@2GHz)

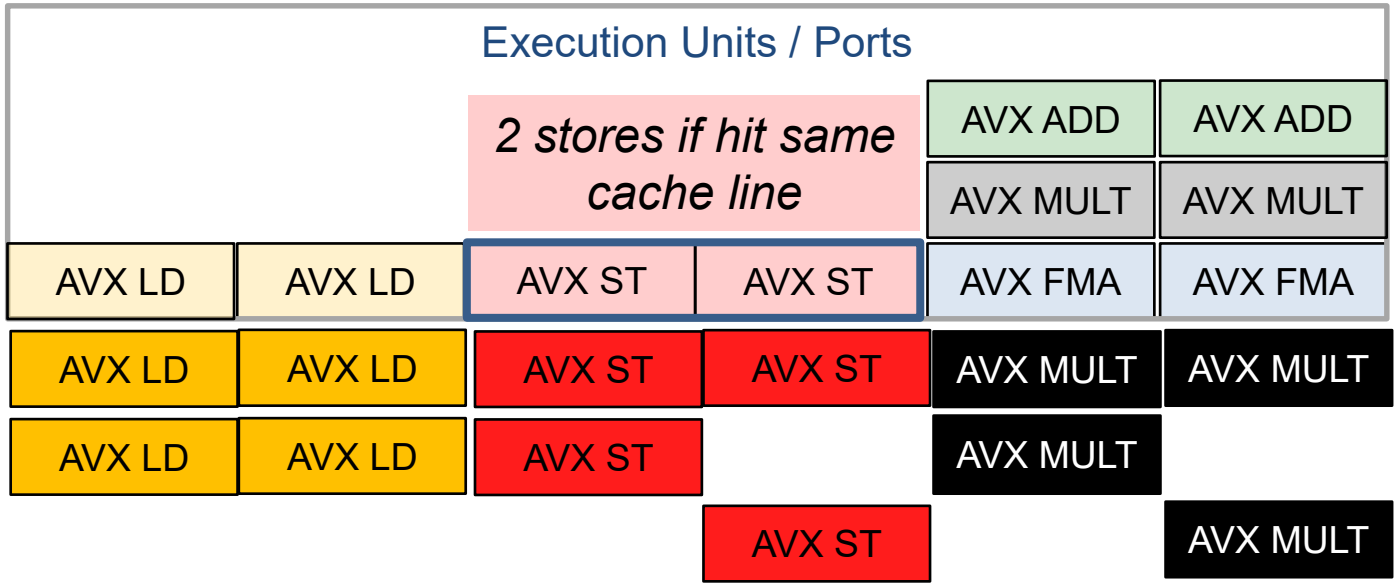
Bottleneck: Instruction throughput

This core has a max. of **5 instructions per cycle** can be executed, i.e. $CPI \geq 0.20 \frac{cy}{inst}$.

Calculating P_{max} - basic machine model / Ice Lake (scalar instr.)

Intel Ice Lake – AVX(2) code: 256-Bit instructions





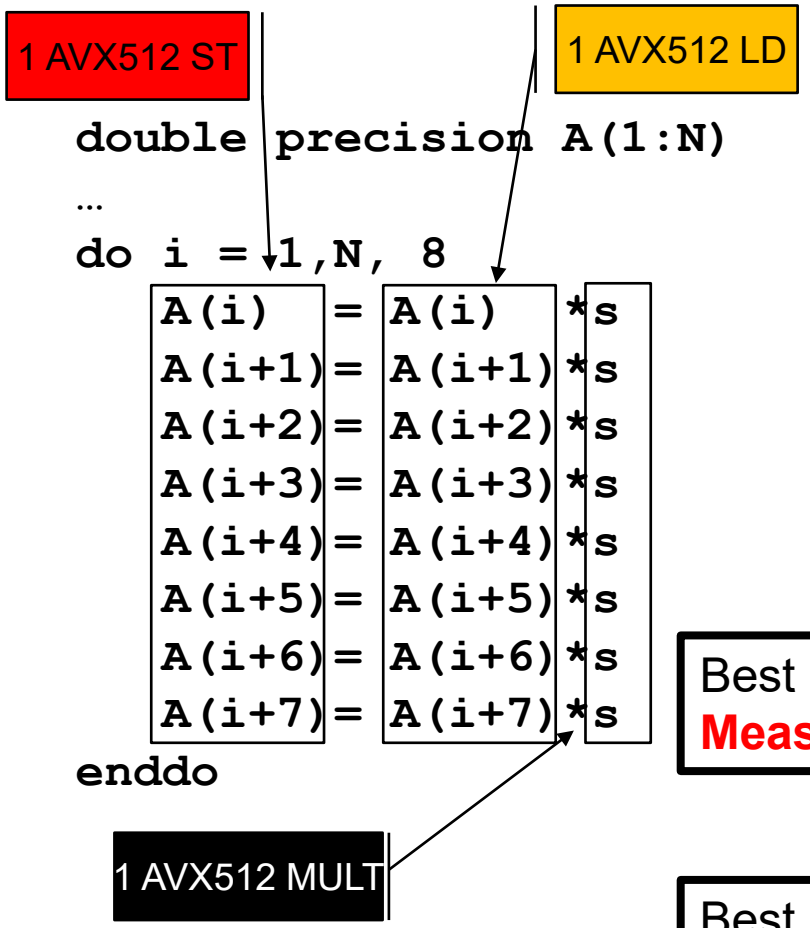
Best performance:	0.6 cy. / AVX MULT	→ 13.3 GF/s (@2GHz)
Measured performance:		→ 12 GF/s (@2GHz)

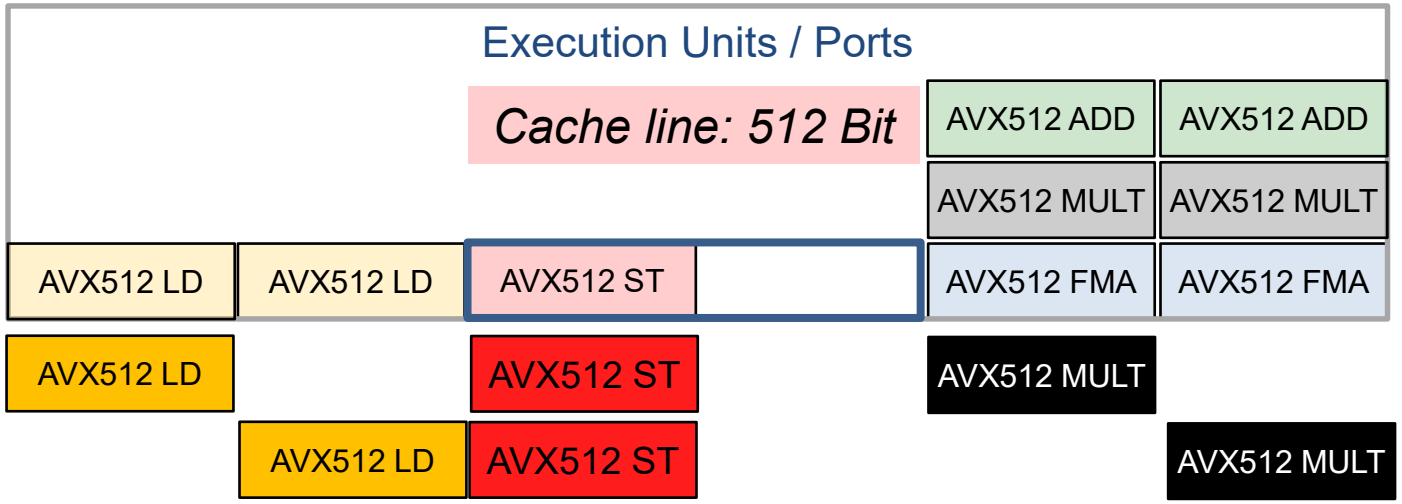
Bottleneck: Instruction throughput

This core has a max. of **5 instructions per cycle** can be executed, i.e. $CPI \geq 0.20 \frac{cy}{inst}$.

Calculating P_{max} - basic machine model / Ice Lake (AVX instr.)

Intel Ice Lake – AVX512 code: 512-Bit instruction





Bottleneck: AVX512 ST

Best performance: 1 AVX512 MULT per cy → 16 GF/s (@2GHz)
Measured performance: → **8.6 GF/s**(@2GHz)

Cascade Lake (same machine model for AVX512)

Best performance: 1 AVX512 MULT per cy → 16 GF/s (@2GHz)
Measured performance: → **13.5 GF/s** (@2GHz)